The FCTOOLS User Manual (Version 1.0)

Amar Bouali, Annie Ressouche, Valérie Roy, Robert de Simone

INRIA

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

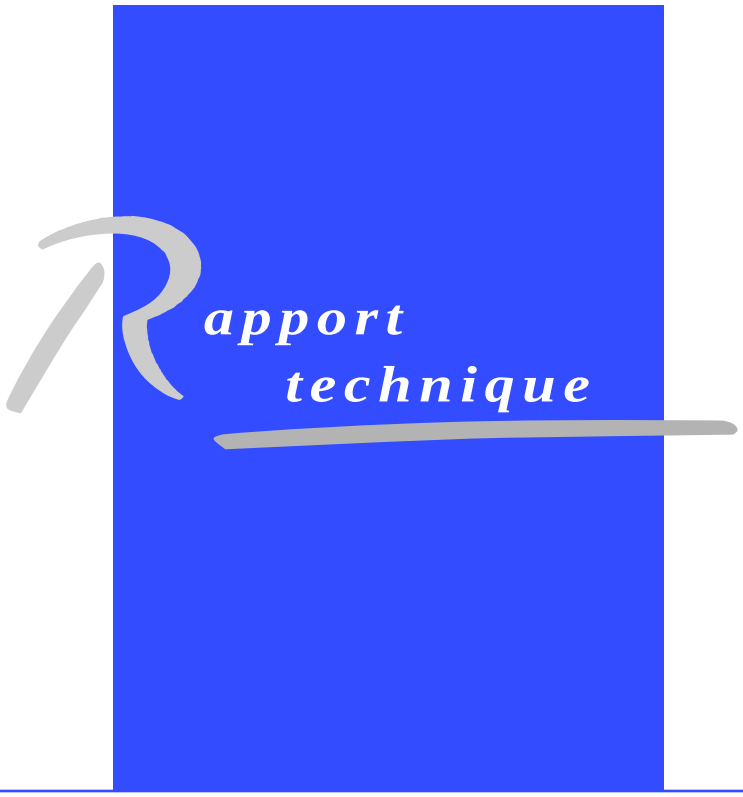
The FCTOOLS *User Manual (Version 1.0)*

Amar Bouali, Annie Ressouche, Valérie Roy, Robert de Simone

N° 191

Avril 1996

THÈME 1



ISSN 0249-6399

The FCTOOLS User Manual (Version 1.0)

Amar Bouali, Annie Ressouche, Valérie Roy, Robert de Simone

Thème 1 — Réseaux et systèmes
Projet Meije

Rapport technique n ° 191 — Avril 1996 — 34 pages

Abstract: We describe a set of modular extensions to our Auto/Graph verification toolset for networks of communicating processes. These software additions operate from a common file exchange format for automata and networks, called FC2. Tool functionalities comprise graphical depiction of objects, global model construction from hierarchical descriptions, various types of model reductions and of verification of simple modal properties by observers, counterexample production and visualisation. We illustrate typical verification sessions conducted on usual academic examples: dining philosophers, mutual exclusion algorithms and round-robin schedulers.

Based on previous experience of drastic state explosion problems we aim here at efficiency in implementation. We use both explicit representation techniques and implicit techniques such as BDDs, with functional overlap at places.

Key-words: Verification tools, networks of communicating processes, automata, algorithms, data structures, BDDs, common format FC2

(Résumé : tsvp)

ENSMP-CMA, B.P. 207 F-06904 Sophia Antipolis cedex
INRIA, B.P. 93 F-06902 Sophia Antipolis cedex

Unité de recherche INRIA Sophia-Antipolis
2004 route des Lucioles, BP 93, 06902 SOPHIA-ANTIPOLIS Cedex (France)
Téléphone : (33) 93 65 77 77 – Télécopie : (33) 93 65 77 65

Manuel de l'utilisateur de FCTOOLS (Version 1.0)

Résumé : On décrit un ensemble d'extensions modulaires à nos systèmes Auto/Graph de vérification des réseaux de processus communicants. Ces nouveaux logiciels opèrent sur la base d'un format commun d'échange pour les automates et les réseaux, appelé FC2. Les fonctionnalités de ces outils comprennent la description graphique d'objets, la construction de modèle global depuis des description hiérarchiques, plusieurs types de réductions de modèle et de vérification de propriétés modales simples par observateurs, la production et la visualisation de contre-exemples. On illustre des sessions de vérification conduites sur des exemples académiques classiques: le problème du dîner des philosophes, des algorithmes d'exclusion mutuels et les schedulers "round-robin".

Basé sur les expériences précédentes se heurtant au problème de l'explosion de l'espace d'états, nous avons pour but ici une implantation efficace. Nous utilisons deux modes de représentation, explicite et implicite par BDDs, en ayant une redondance des fonctionnalités dans chaque mode.

Mots-clé : Outils de vérification, réseaux de processus communicants, automates, algorithmes, structures de données, BDDs, format commun FC2

1 Presentation

Systems of communicating and synchronising entities are usually hard to specify in a correct fashion, due to problems of distributed control and parallelism. In the last decade a number of verification softwares were implemented to provide computer assistance in the design and correctness checking of such system descriptions, and used to study distributed algorithms, protocols and embedded systems. Most commonly these toolsets are based on finite state modeling of underlying global configurations, and graph-theoretic algorithms.

Our pioneering AUTO/GRAPH toolset was exploring the power of so-called "proof-by-reduction" techniques, where methods for compositional reductions of finite state structures try to suppress as much as possible the combinatorial explosion problem. Functions such as state quotient (with respect to behavioural equivalences), behavioural abstraction or context filtering were at the heart of the system, in addition to graphical or textual *process algebraic* hierarchical description facilities, and other practical auxiliary functions.

The present *User Manual* describes basically the "next generation" AUTO/GRAPH implementation. Decision for this reimplementation was based on a number of facts. First, as functionalities were progressively added the old implementation grew larger and harder to maintain; the new one had to be modular, consisting in a set of carefully chosen functions which could be combined together for efficient verification. Second,due to national and international collaborative projects we wanted the new toolset to be open for joint use with other "foreign" verification tools, which could nicely complement its functionalities; a "low-level" file exchange format (covering automata and hierarchical networks of such) called FC2 was then designed, and used in particular in between various software modules. Last, new *symbolic* techniques for implicit representation of finite state machines by so-called *Binary Decision Diagrams* had appeared, and were becoming prominent in the neighboring domain of *synchronous reactive systems* (real-time systems and synchronous hardware for instance). We adapted our verification techniques to this type of implementation structures and the relevant algorithmic style, in the scope of asynchronous processes communicating by rendez-vous synchronisation.

The result is a new set of construction/reduction/analysis/diagnostics functions, corresponding to a number of UNIX commands working from and to FC2 files. The three main software modules are: AUTOGRAPH, for graphical edition and display; FC2EXPLICIT, for manipulation of enumerated finite state machines; FC2IMPLICIT, for manipulation of symbolic finite state machines. Each fulfills several distinct functions, sometimes with redundancy between FC2EXPLICIT and FC2IMPLICIT. Other auxiliary modules exist as well.

By nature FCTOOLS is in perpetual ongoing expansion, as more useful analysis functions are identified and characterised as efficient algorithms. This manual describes only the current state, which may already be obsolete by the time of reading in case a next version is already out. Information on system availability and documentation can be obtained on request from *fc2team@cma.cma.fr*, or from URL *http://cma.cma.fr/Verification/verif-eng.html*.

The next section describes the overall architecture of software modules comprised in the toolset, with an informal description of their individual functionalities and how they can be combined. Then a working description of `Unix` commands and options is given, followed by

a small session example. Each verification module is then further presented and explained, with insights on its internal algorithms, and indications on how-to-use for best efficiency.

2 Modular Software Architecture

The verification toolset comprises a number of stand-alone tools, each implementing some well-defined functionalities. Tools may be used in succession through the common FC2 file description format. At a deeper programming level, most of our tools use identical internal representation (in terms of C++ *classes*), so that combination of code is also possible there. See the appended *Implementation Manual* for details.

Figure 1 sketches the overall software architecture, with tools/functions figured in oval shapes and objects/data in rectangular frames. Explicit mention is made to FC2 format where available for objects (for instance, there is no direct representation of BDDs in FC2).

In the sequel we present the FC2 format and the individual verification tools at very abstract level. Each tool will be extensively presented later on.

2.1 The FC2 format

The FC2 format was originally designed to interface several preexisting verification tools. In this way these heterogeneous tools could be further developed independently, while used in cooperation for their complementary features.

The format allows for description of labeled transition systems and networks of such. While the format is not "syntax-friendly" (as it represent objects which are supposedly obtained by translation or compilation), it is still reasonably natural: automata are tables of states, states being each in turn a table of outgoing transitions with target indexes; networks are vectors of references to subcomponents (i.e., to other tables), together with synchronisation vectors (legible combinations of subcomponent behaviours acting in synchronised fashion). Subcomponents can be networks themselves, allowing hierarchical descriptions.

In addition a permissive labeling discipline allows a variety of annotations on all distinct elements: states, transitions, automata and networks as a whole. It is through this labeling that *behavi*oural action labels are provided of course, but also *struct*ural information for source code retrieval, *logic*al model-checking annotation and even private *hook*ed informations. Processes augmented with time, value or probability informations could certainly benefit from that, and this is not limitative. Annotative labels are dealt with as regularly as possible in syntax, in simple form at predictable location, so that they can be treated smoothly at parsing time by any tool, often by simply disregarding them if they do not address the tool's specific functionalities. The actual labeling contents are stored in tables forming the objects headers, so that only integers references to table entries are actually present in the object bodies themselves (automata or networks). Finally, labels can be structured by simple operators (*sum*, *product* and several others) to allow richer information.

More about the FC2 format can be found in [3].

2.2 Functional Modules

A typical case-study analysis will contain a number of typical design steps, corresponding to successive application of distinct *functional modules* from our toolset. The main such functions are:

description of the network of communicating agents (possibly graphically) The graphical editor AUTOGRAPH allows to draw such descriptions much in the usual fashion of process-algebraic terms, and then produces FC2 format representations. It also contains the annotation labeling facilities. See autograph description in this manual for details.

linking of multifile descriptions Large hierarchical system descriptions can be split between different files (for instance as different AUTOGRAPH windows). The tabulated naming informations in resulting FC22 files need not be consistent across files, and so merging these partial descriptions into a single file for later analysis takes some bookkeeping care.

construction of "some form of" global model Model-based automatic verification relies on expansion of network into a global state-transition model. Two main implementation techniques can be used here: the *extensional approach* with a classical representation of expanded automata with enumerated states and transitions; the *symbolic* approach, based on implicit representation by *Binary Decision Diagrams* of sets of states (only), while representation of the *full* transition relation is avoided, and remain parted by possible events, somehow in the Petri net fashion. Our tools cover both modes of implementation with large mutual redundancy, so that best efficiency can be thought according to each given specification.

Of course global models can suffer state or bdd size explosion problems, leading to the well-known bottleneck of the approach. Several methods can be used to refrain this explosion, like abstracting or minimizing (explicit) subnetworks at intermediate level of hierarchical descriptions. In all cases the global model expansion remains a fundamental part of verification systems, even if applied in particular settings or on transformed objects to cope with complexity.

reduction/abstraction of the model Smaller models can be obtained in roughly two ways. First, one can *abstract* the actual concrete *behaviours* into new ones of a more concise nature; it corresponds to the converse of *action refinement*, where more behavioural details are progressively added (here they are abstracted away). Second, states with equivalent potential behaviours can be merged (using bisimulation for instance). Note that behaviour abstraction paves the way to state reduction, as it usually removes differences between otherwise similar states (consider for instance *observational* behaviours, including *tau* invisible steps inside visible ones).

These techniques can be even more beneficial when applied in a compositional fashion, minimising intermediate level descriptions.

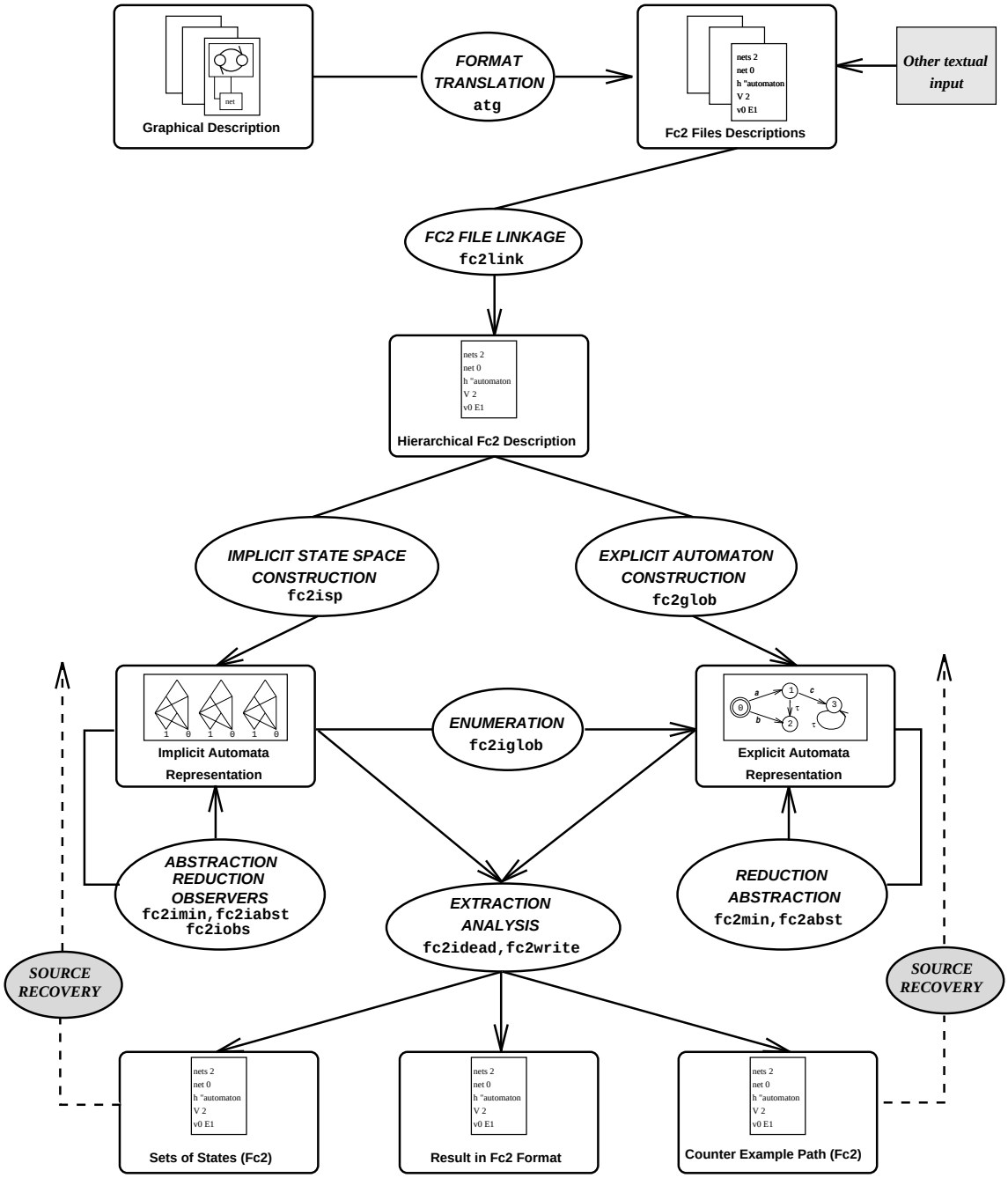


Figure 1: Software hierarchy

Another way of reducing the model is by taking into consideration a given context limiting the state-space exploration. This context can for instance be extracted from a given property to check.

specification of properties and model-checking There are several ways of specifying correctness properties. Some basic obvious properties can be stated directly as characteristics of the finite state model, and checked by simple analysis on it: existence of *deadlock*, *livelock* or *divergent* states for instance. More refined properties can be expressed either as *modal temporal logic* formulae or as *specification automata*. Distinctions are usually made according to visions of time: in *linear time* frameworks properties of behavioural sequences are considered, while in *arborescent branching time* frameworks one gets interested in properties of states through their past and future neighbours. An abundant literature was devoted to comparison of expressiveness and design of algorithmic methods best adapted in various cases. Our tools focus on specification of properties as specification automata, given that the temporal logic approach seemed well treated elsewhere.

Again, there are two approaches to compare two finite state models, one being the specification of some (maybe partial) intended behaviour of the other. The first one is *bisimulation comparison*; it works well when "partial" means "abstract", when time is "branching" and the processes may both exhibit *nondeterministic* behaviours. The second one considers the specification automaton as an *observer*, and performs some kind of product machine construction to deduce whether (un)desirable joint configurations can be attained; this approach, known as "on-the-fly" technique, works well under determinism assumptions on the specification automaton. Also, as a rule of thumb, "explicit representation" methods win in the first approach, while "implicit representation" are best suited to the second one.

Another dimension to the property specification problem depends on whether the analysed process is viewed as a *transparent* or a *black* box, that is whether the property may explicitly refer to control points (states) in it, or only through behavioural abilities (leading to or possible from the states in question). For instance a mutual exclusion property can most naturally be stated by the fact that no global configuration may contain specific local states in parallel subcomponents. Thus the toolset will have to provide ways of composing this type of property from the system description, and this *without affecting the latter* for each property to prove.

counterexample production at the network level Diagnostics from analysis and model-checking on incorrect descriptions usually result in either sets of (undesirable) states, or counterexample paths. Typically, *deadlock* or *divergent* states are of the first form, while runs without bisimilar counterpart are of the second form.

With the addition of prior reduction phases these results are produced on smaller automata, and are themselves usually smaller than the corresponding ones on original networks. But these now have to be retrieved, if the user is to be informed at a level of

description he/she can understand. The `struct` annotation field of the FC2 format was in fact used to carry exactly that minimal information which allows reconstruction. For instance, if weak bisimulation minimisation was used and hidden transitions thus removed, these transitory behaviours may have to be rediscovered to glue actual visible steps back together.

Diagnostic reconstruction may be a time penalty, but is only necessary in case of property failure, and avoids storing much extra information at all times, which could abort verification for lack of space.

Figure 1 displays our global software architecture, with tool names and functionalities and types of arguments and results. Next section will provide a synthetic overview of each tool and ways to use it in practice.

2.3 Tools and Commands

We now describe the different software modules at the level of UNIX commands, with names and options.

Remark: most of the transformation tools generate single FC2 description, dumped on screen (UNIX standard output). In order to save the result in a file, one has to redirect the output of the command to that file.

- `atg`:

  SYNOPSIS:

  UNIX command for AUTOGRAPH, the graphical editor and display system for FC2 descriptions. AUTOGRAPH uses usual process algebra conventions for graphical representation of automata and networks, and provides translation into FC2 format. AUTOGRAPH currently reads only plain automata from this format, while a dedicated `.atg` file format can be loaded and written on file for any drawing, even ill-structured or incomplete.

  USAGE:

  `atg` *[files.fc2][files.atg]*

  RESULT:

  A menu bar for graphical edition and a specific window for each loaded file (from `.fc2` automata only initial states are displayed at first). AUTOGRAPH and its functionalities are further described in section 3.

- `fc2link`:

  SYNOPSIS:

  Linker of (partial) FC2 files produced by ATG. It redirects references to a subcomponents to its actual description (found from another file), and matches the labeling indexes.

  USAGE:

  `fc2link -main [-nodebug]` *file.fc2* `[`*file1.fc2...*`[`*fileN.fc2*`]...]`

  RESULT:

  The result is a single FC2 file containing the complete hierarchical FC2 description of `net0` in file *file.fc2* together with all its subcomponents found in any file mentioned. Default resulting file contains verification debugging information used by source recovery functions, such as the file names of individual FC2 components given under an FC2 expression recalling the hierarchy of the network. This extra information can be discarded from the result by setting the `-nodebug` option.

  Misformed descriptions end up in so-called "consistency errors". The result is output on screen.

- `fc2min`:

  SYNOPSIS:

  (Explicit) Automata minimizer with respect to strong, weak and branching bisimulation.

  USAGE[1]:

  `fc2min` *-bisimulation* `[-fc2] [-debug]` *file.fc2*

  The option *bisimulation* can be one of the options `s, w` or `b` for strong, weak and branching bisimulation respectively.

  RESULT:

  If option `-fc2` is set, the result is the quotient automaton in FC2 format. Otherwise it is a partition of the state space into equivalence classes. The `-debug` source recovery option adds, for each quotient state or partition element, a description of its content as sum (union) of state references from the initial automaton. This information is stored in the `struct` field of the new states in the FC2 structure.

- `fc2implicit`:

  SYNOPSIS:

  Symbolic manipulation of labeled synchronized automata vectors (FC2 networks). It contains several functionalities, selected by options.

  USAGE: The command can be invoked with either one or two argument files:

  1. One file mode:

     `fc2implicit [-reach | -dead | -live | -dive]`
     `[-s | -w | -b [-itau]] [-debug] [-fc2]` *file.fc2*

     where

     `-reach`: computes the set of reachable global states.

     `-dead, -live, -dive`: computes the set of deadlock, livelock and divergent global states of the network respectively. If option `-fc2` is set in addition, `fc2implicit` generates a counterexample path in FC2 (as a string automaton), leading from the initial state to one of the computed states.

     `-s, -w, -b`: computes the strong, weak and branching equivalence partition respectively. If option `-fc2` is set, then generates an FC2 description of the quotient automaton. Option `-itau` can be added for branching bisimulation to turn off the τ-closure memorization, replaced by a local recomputation at need.

     `-debug`: adds extra information for source recovery in the `struct`labels of global nets, states and transitions.

[1] *file.fc2 must contain a single automaton. Otherwise, an error message is generated. If minimization is asked for the global automaton of a network described in a fc2 file,* `fc2explicit/fc2implicit` *processors should be used instead.*

2. Two files mode:

   `fc2implicit {-seq | -weq } [-debug] [-fc2]` *file1.fc2 file2.fc2*

   where

   `-seq, -weq`: performs the strong and weak bisimulation comparison between the topmost nets of both files.

   `-debug`: produces a counterexample path in FC2 leading to a state without equivalent in the other automaton, with other infos (iteration level in the partitioning, ...).

SHORTHAND COMMANDS:

The following UNIX commands are equivalent to the general `fc2implicit` command with particular options. The `i` letter following `fc2` here stands for `implicit`.

```
fc2ireach   =  fc2implicit -reach
fc2idead    =  fc2implicit -dead -fc2
fc2ilive    =  fc2implicit -live -fc2
fc2idive    =  fc2implicit -dive -fc2
fc2istrong  =  fc2implicit -s
fc2iweak    =  fc2implicit -w
fc2ibranch  =  fc2implicit -b
fc2iglob    =  fc2implicit -reach -fc2
```

RESULT :

Whenever option `-fc2` is set, generates an FC2 description of the result. Otherwise produces information messages (result size, existence of deadlocks for instance).

- `fc2explicit`

  SYNOPSIS:

  Explicit manipulation of labeled synchronized automata vectors (FC2 networks). It contains several functionalities, selected by options.

  USAGE: The command can be invoked with either one or two argument files. Currently only the `-abstract` option uses two files.

  `fc2explicit [-s | -w | -b | -abstract] [-comp | -global] [-bitset] [-fc2] [-debug] [-o file.fc2]` *file1.fc2* [*file2.fc2*]

  where

  `-abstract`: Assumes one file contains a net description and the other an abstraction criterion. Performs the abstraction of the global automaton of net w.r.t. the abstraction criterion. Further description of abstraction use can be found in section 7.

  `-seq, -weq`: Requires two FC2 files containing two networks. Performs the comparison of the two systems with respect to strong (`-seq`) or weak (`-weq`) bisimulation. In case of non equivalence potential states without match are searched for as early as possible, and a path leading to such a state is provided as result.

  `-comp`: Computes the global automaton from the network contained in the argument file in a compositional way, following the hierarchical description in nested subnets. Used in conjunction with `-s, -w, -b` options to alternate minimisation and construction phases.

  `-global`: Computes the global automaton from the network contained in the file argument in its "flattened" version (non hierarchical). Default value.

  `-s, -w, -b` Applies strong, weak or branching bisimulation minimisation on network contained in file argument. Can be combined with `-comp` option. Internally invokes `fc2min` (see above) on each intermediate automaton.

  `-bitset` Computes the state space by applying action events under a bitset scheme algorithm for replacement of local states in the vector. Used best with the `-global` option, on large vectors of small individual automata components. See further FC2EXPLICIT description in 5.1.

  `-o`: provides a filename for output.

  `-fc2`: if set, result is the FC2 description of the quotient automaton; otherwise only size figures are printed. Prints on standard output, except if `-o` option is used.

  `-debug`: if set, automata states are decorated with structure information for source recovery on original network description.

SHORTHAND COMMANDS:

The following UNIX commands are equivalent to the general `fc2explicit` command with particular options.

`fc2glob`	=	`fc2explicit -global -fc2`
`fc2strong`	=	`fc2explicit -global -s -fc2`
`fc2weak`	=	`fc2explicit -global -w -fc2`
`fc2branch`	=	`fc2explicit -global -b -fc2`
`fc2compstrong`	=	`fc2explicit -comp -s -fc2`
`fc2compweak`	=	`fc2explicit -comp -w -fc2`
`fc2compbranch`	=	`fc2explicit -comp -b -fc2`
`fc2abst`	=	`fc2explicit -abstract -fc2`
`fc2abststrong`	=	`fc2explicit -abstract -s -fc2`
`fc2abstweak`	=	`fc2explicit -abstract -w -fc2`
`fc2abstbranch`	=	`fc2explicit -abstract -b -fc2`

RESULT :

Whenever option `-fc2` is set, generates an FC2 description of the result. Otherwise produces information messages (result size for instance).

- `fc2view`

  SYNOPSIS :

  Source recovery viewer. When a path is given as argument (the path must be retrieved from a global automaton of a network), FC2VIEW pops up two windows, one containing the graphical tree representing the hierarchy of nets forming the network from which the path has been recovered, and a control panel to simulate the path. Nodes and leaves of the tree are labeled by the names of the corresponding nets. In the control panel, buttons are provided to fire transitions in the path going back and forth, and step by step, plus a graphical scale allowing the user to access directly at some depth in the path and fire the transition at that depth. Each time a transition is fired, its (global) label is displayed in a dedicated zone (near the name of the path) and so are in the graphical tree the local ones that have produce the global label: these labels are displayed in the graphical tree, near the components that have offered them, which are themselves highlighted. An extra feature allows the user to visualise the FC2 description of any net appearing in the tree by clicking on its displayed name. Actives part (if any) of the text are also highlighted (source and target states of current active transition) as well as the text background when the component is active.

  USAGE :

  `fc2view` *file.fc2*[2]

  where *file.fc2* is assumed to contain a path synthesized from a network using the `-debug` option, so that it can be displayed as a distributed run on the range of corresponding FC2 files. Creates as many (slave) windows as there are automata components in the network, in their FC2 syntax. Each window displays current local share of transition in a graphical header, and FC2 text below on demand. Simulation can travel back and forth under control of a graphical panel.

  RESULT :

  See above

- `fc2hide`

  SYNOPSIS :

  When a FC2 network is given in input, pops up a window showing the list of action labels the network can perform at the global level. Mouse clicking on labels permits selection of labels to hide. When selection is finished, the user can save the result in a new FC2 file as a new network where the selected global label has been renamed into the silent action τ. This allows to restrict the range of visible behaviours, and thus to increase observational reduction.

[2] *The argument file must contain a single string automaton containing a path (obtained by* `fc2idead` *for instance), and containing debug informations*

USAGE :

`fc2hide` *file.fc2*

Assumes *file.fc2* contains a network.

RESULT :

A new network where selected labels of synchronisation vectors of the main net are renamed into τ.

2.4 First steps: a session example

We now illustrate the basic verification features on the famous *dining philosophers* problem. More advanced features will be demonstrated later on.

The graphical ATG description of the system (in the case of 3 philosophers) is displayed in figure 2 (in its Postscript output form). It consists essentially of the automata describing the possible behaviours of the forks and of halfbrains for philosophers. A full philosopher is obtained by synchronising these halves on `eat`ing and `think`ing (each half deals with one fork). The full synchronisation network is also displayed, with visible actions becoming indexed by a philosopher's rank.

We now suppose these three parts (the fork, halfbrain automata and the network) have been translated (by ATG) into distinct FC2 files, say `fork.fc2`, `halfbrain.fc2` and `philonet.fc2`. The FC2 version of the fork automaton is displayed in figure 3. The partial description of the network, with only component interface declaration for the fork and halfbrain, is displayed in figure 4.

Linking these files will produce the appropriate correspondence between these "subsystem calls" and their automata contents from the other files.

```
0-duick$ fc2link -main philonet.fc2 fork.fc2 halfbrain.fc2 > philo.fc2
-- fc2link: education version v0
-- fc2tool: parsing fc2 file: philonet.fc2.
-- fc2tool: file: philonet.fc2 parsed successfully
-- fc2tool: parsing fc2 file: fork.fc2.
-- fc2tool: file: fork.fc2 parsed successfully
-- fc2tool: parsing fc2 file: halfbrain.fc2.
-- fc2tool: file: halfbrain.fc2 parsed successfully
-- fc2link: File "philonet.fc2"
-- fc2link: net number 0 has struct "philonet"
-- fc2link: net number 1 has struct "fork"
-- fc2link: net number 2 has struct "halfbrain"
-- fc2link: File "fork.fc2"
-- fc2link: net number 0 has struct "fork"
-- fc2link: File "halfbrain.fc2"
-- fc2link: net number 0 has struct "halfbrain"
-- fc2link: Check consistency on class of net 0, file philonet
-- fc2link: Check consistency on class of net 0, file fork
-- fc2link: Check consistency on class of net 0, file halfbrain>
0-duick$
```

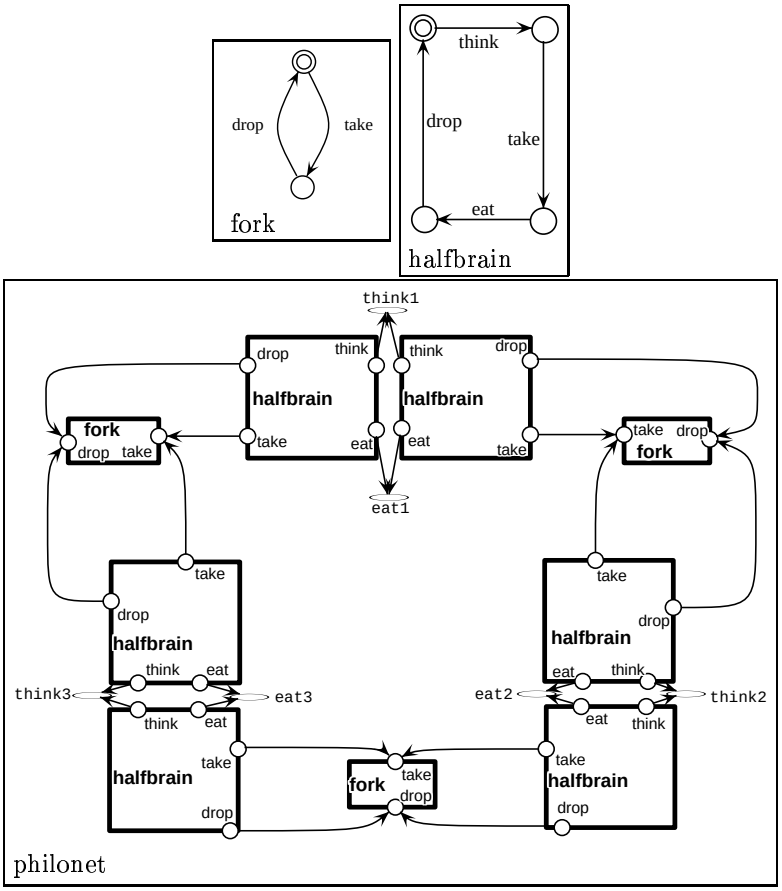


Figure 2: The 3 dining philosophers specification

The result is displayed in figure 5
Now the description can be submitted to our analysis and verification tools.

2.4.1 Implicit evaluation of the global system

We first evaluate the global system to have an idea of the size of the state space. We use for that symbolic methods based on BDDs that allow easy evaluation of global state spaces.

```
0-duick$ fc2implicit -reach philo.fc2
-- fc2implicit: education version v0
```

```
nets 1
hook"main" > 0
struct"fork"
net 0
behavs 2
:0 "take"
:1 "drop"
logic "initial">0
hook "automaton"
vertice 2
vertex0
edges 1
edge0
behav 0
 -> 1
vertex1
edges 1
edge0
behav 1
 -> 0
```

Figure 3: file `fork.fc2`

```
-- fc2tool: parsing fc2 file: philo.fc2.
-- fc2tool: file: philo.fc2 parsed successfully
-- fc2implicit: Making reachable state space
-- fc2implicit: Reachable states: «214» - BDD nodes: «85»
0-duick$
```

The global automaton has 214 states. The BDD that represents it has 85 nodes only.

2.4.2 Finding and Recovering the Deadlocks

This academical problem is known to have deadlocks. We have a way to detect them and to extract an example path leading to a deadlock from the global initial state. Here is the session using `fc2implicit`:

```
0-duick$ fc2implicit -dead -fc2 philo.fc2 > deadpath.fc2
-- fc2implicit: education version v0
-- fc2tool: parsing fc2 file: philo.fc2.
-- fc2tool: file: philo.fc2 parsed successfully
-- fc2implicit: Making reachable state space
-- fc2implicit: State space depth: 13
-- fc2implicit: First deadlock(s) detected at depth 7
-- fc2implicit: Reachable states: «214» - BDD nodes: «85»
-- fc2implicit: Global automaton has 2 DEADLOCKS state(s) - BDD nodes:
«27»
0-duick$
```

The first detected deadlocks have been found at depth 7 in the global automaton, that is the shortest path leading to a deadlock has 7 states and 6 transitions. As we have set the option `-fc2`, an example path has been extracted and written in FC2 in `deadpath.fc2`.

```
nets 3
hook"main" > 0
struct"philonet"
net 1
structs 1
:0 "fork"
behavs 2
:0 "take"
:1 "drop"
struct 0
behav 1+0
hook "synch_vector"
net 2
structs 1
:0 "halfbrain"
behavs 4
:0 "eat"
:1 "take"
:2 "drop"
:3 "think"
struct 0
behav 2+1+0+3
hook "synch_vector"
net 0
behavs 6
:0 "eat1"
:1 "eat2"
:2 "eat3"
:3 "think1"
:4 "think2"
:5 "think3"
struct _< 1,2,2,2,1,2,2,2,1

hook "synch_vector"
vertice 1
vertex 0
edges 18
edge 0
behav 3 < *,*,*,3,*,3,*,*,* ->0
edge 1
behav 0 < *,*,*,0,*,0,*,*,* ->0
edge 2
behav 5 < *,3,3,*,*,*,*,*,* ->0
edge 3
behav 2 < *,0,0,*,*,*,*,*,* ->0
edge 4
behav 1 < *,*,*,*,*,*,0,0,* ->0
edge 5
behav 4 < *,*,*,*,*,*,3,3,* ->0
edge 6
behav tau < *,*,*,*,*,2,*,*,1 ->0
edge 7
behav tau < *,*,*,*,*,1,*,*,0 ->0
edge 8
behav tau < 1,*,*,2,*,*,*,*,* ->0
edge 9
behav tau < 0,*,*,1,*,*,*,*,* ->0
edge 10
behav tau < 1,*,2,*,*,*,*,*,* ->0
edge 11
behav tau < 0,*,1,*,*,*,*,*,* ->0
edge 12
behav tau < *,2,*,*,1,*,*,*,* ->0
edge 13
behav tau < *,1,*,*,0,*,*,*,* ->0
edge 14
behav tau < *,*,*,*,*,*,2,*,1 ->0
edge 15
behav tau < *,*,*,*,*,*,1,*,0 ->0
edge 16
behav tau < *,*,*,*,1,*,*,2,* ->0
edge 17
behav tau < *,*,*,*,0,*,*,1,* ->0
```

Figure 4: file `philonet.fc2`

```
% FC2 file generated by fc2link from FC2 files:
% philonet.fc2 (main) fork.fc2 halfbrain.fc2

D
prefix file(any any) -> any

nets 3
h "main">0
s file("philonet",0) < file("fork",0),file("halfbrain",0),file("halfbrain",0),
file("halfbrain",0),file("fork",0),file("halfbrain",0),file("halfbrain",0),
file("halfbrain",0),file("fork",0)

net 1
B2
:0 "take"
:1 "drop"

s "fork" l "initial">0 h "automaton"
V2
v0 E1
e0 b 0 r 1
v1 E1
e0 b 1 r 0
net 2
B4
:0 "eat"
:1 "take"
:2 "drop"
:3 "think"

s "halfbrain" l "initial">0 h "automaton"
V4
v0 E1
e0 b 3 r 1
v1 E1
e0 b 1 r 2
v2 E1
e0 b 0 r 3
v3 E1
e0 b 2 r 0
net 0
B6
:0 "eat1"
:1 "eat2"
:2 "eat3"
:3 "think1"
:4 "think2"
:5 "think3"

s "philonet"<1,2,2,2,1,2,2,2,1 h "synch_vector"
V1
v0 E18
e0 b 3<*,*,*,3,*,3,*,*,* r 0
e1 b 0<*,*,*,0,*,0,*,*,* r 0
e2 b 5<*,3,3,*,*,*,*,*,* r 0
e3 b 2<*,0,0,*,*,*,*,*,* r 0
e4 b 1<*,*,*,*,*,*,0,0,* r 0
e5 b 4<*,*,*,*,*,*,3,3,* r 0
e6 b tau<*,*,*,*,*,2,*,*,1 r 0
e7 b tau<*,*,*,*,*,1,*,*,0 r 0
e8 b tau<1,*,*,2,*,*,*,*,* r 0
e9 b tau<0,*,*,1,*,*,*,*,* r 0
e10 b tau<1,*,2,*,*,*,*,*,* r 0
e11 b tau<0,*,1,*,*,*,*,*,* r 0
e12 b tau<*,2,*,*,1,*,*,*,* r 0
e13 b tau<*,1,*,*,0,*,*,*,* r 0
e14 b tau<*,*,*,*,*,*,2,*,1 r 0
e15 b tau<*,*,*,*,*,*,1,*,0 r 0
e16 b tau<*,*,*,*,1,*,*,2,* r 0
e17 b tau<*,*,*,*,0,*,*,1,* r 0
```

Figure 5: The 3 philosophers in FC2 format

Now we visualize back in ATG this result that we picture out in figure 6.

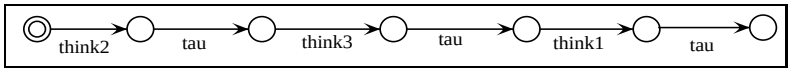


Figure 6: A deadlock path

The deadlock corresponds to the case where each philosopher takes a fork (the τ action after each `think` action): then no action can be further enabled from any of them.

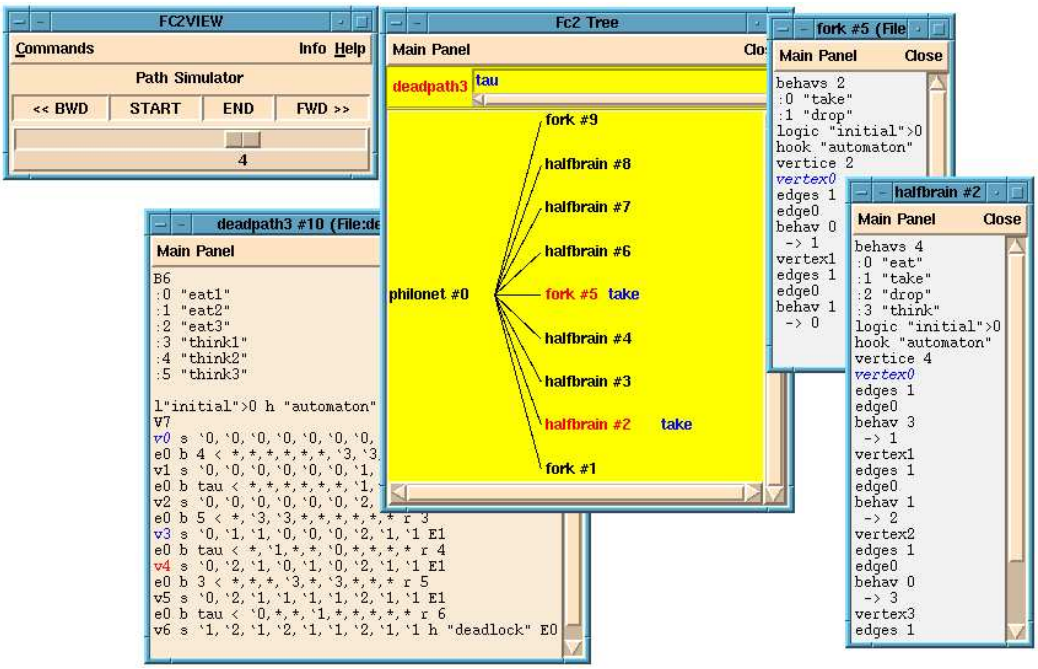


Figure 7: FC2VIEW display

Now if `-debug` option was added to the FC2IMPLICIT command, further annotations were appended to the path example so as to allow source recovery. Then the path can be simulated as a run on FC2 files using FC2VIEW, or even visualised graphically on an original displayed network with AUTOGRAPH.In the former case, a graphical view of the network tree is displayed with a control panel that allows the user simulate the path: active components are highlighted on the way as well as the action label they perform. A specific zone is dedicated for the path name and the global realised action name: FC2 textual description of the path and components can be displayed by clicking on appropriate names. Active component's text is also highlighted as well as active source and target states (see figure 7).

In the latter case one needs only load the path in FC2 to AUTOGRAPH, and then selects the Debug:Edge button from the menu bar. Then each selection of an edge will highlight the source and target states at all components in their respective AUTOGRAPH windows, and active communications at ports in the synchronisation network (see figure 8).

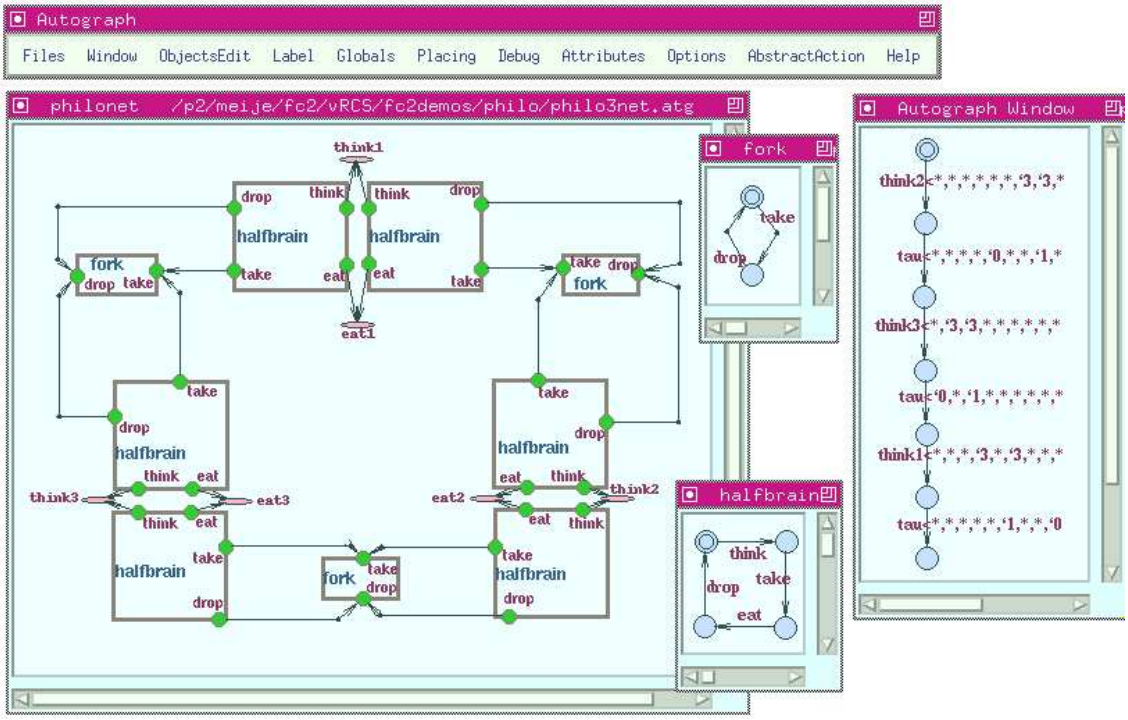


Figure 8: AUTOGRAPH display

3 The Graphical Editor AUTOGRAPH

AUTOGRAPH (invoked under the UNIX command `atg` under `X-windows`) is a graphical display system for both labeled transition graphs and networks of communicating systems. Lay-out is very much in the tradition of process algebra graphical depiction, as shown in figure 8. Objects in AUTOGRAPH can also be extensively annotated so as to match the FC2 format standards. In section 2.4, figure 2 was produced from AUTOGRAPH graphical displays.

AUTOGRAPH can be used to graphically edit systems but also to visualise automata that were produced elsewhere, typically as an output of verification. Then when reading an FC2 file AUTOGRAPH prompts the user for interactive unfolding and positioning of successive states. An automaton can also be automatically drawn (using a spring-like attraction/repulsion algorithm between states). Visualisation of networks is under construction, as is visualisation of counterexample runs on existing networks.

3.1 General Features

In practice AUTOGRAPH is a multi-window, unstructured editor: system descriptions are checked for structural coherency only at translation into FC2 format, and subsystem parts contained in different windows are translated independently in separate files and *not* linked together. This allows the user freedom to work temporarily with incomplete descriptions, and to reuse system parts in various compositions. Therefore AUTOGRAPH is based on two file representation formats: FC2 for structured objects, and ATG for possibly inconsistent drawing descriptions, containing additional graphical positioning data.

3.1.1 Menu Bar

AUTOGRAPH fronts the user with a single menu bar, from which all editing functions applicable to all graphical windows are selected. As a result some functions may need an extra mouse click in the window(s) to be concerned (like in the *Save to File* function). The *Files*, *Windows*, *ObjectsEdit* and *Labels* menus deal with management of the respective types of objects. While rather self-explanatory they are described in more details in the sequel. The *Globals* menu deals basically with cut-and-paste and miscellaneous functions to be applied undistinctively on all editable objects. *Placing* deals with positioning of folded objects, and *Attributes* allows to play with fonts and colors. The *Abstract Action* menu deals with edition of an automaton representing an abstract criterion. The *Help* menu contains useful information on how to use ATG.

3.1.2 Mouse Buttons

The three mouse buttons are **different** bindings: the functions selected from menus have to be applied using the left mouse button, while the middle button moves any kind of objects, and the right button (pre)selects a number of objects, or all objects in a given rectangular zone, typically to be applied the next function as a whole.

3.1.3 Editable objects

Consisting of graphical editable objects AUTOGRAPH offers *vertices* for states, *boxes* for subsystems, *ports* for signal interface, *edges* for both automata transitions and port connections, and *"webs"* for multipoint extended connections. All such objects can be annotated with semantic informations as allowed in the FC2 format. Behavioural labeling of automata transitions form their action abilities as usual. The only structural requirement of autograph is that ports only occur on boxes and edges in between vertices, ports and webs altogether (no free end to an edge).

3.2 File Management

This menu contains in addition the `quit` menu button.

AUTOGRAPH saves files in `.atg`, `.fc2` or .ps formats. Postscript format is not scaled to fit (a given page size).

AUTOGRAPH reloads files from `.atg` format, and reads from `.fc2` format in case the file contains a single automaton (in the current version). In the second case the user must unfold successive states to provide the actual lay-out. At first only the initial state is pictured. Then, by dragging a phantom line to any point in the drawing zone the user indicates both a main direction and a minimal distance from which to place new vertices.

3.3 Window management

Windows can be created and deleted from the corresponding menu. In addition they can be resized to fit the actual drawing, or given a *title* name. Such names are important as they will become the FC2 name of the window content (network or automaton).

In general drawings may exceed the window size (with usual scrollbar facilities). The *Window:See/Hide Global* menu button allows to pop up a global view spanning the whole object. Such windows cannot be edited, but unexplored vertices can easily be spotted from their highlighting, and the regular view from the editable window can be repositioned by its phantom.

Each window keeps the memory of its last operation, which can be undone by the *Window:Undo* button.

3.4 Edition

Objects can be edited from general functions in the `ObjectsEdit` menu. Shorthands keyboard bindings allow fast selection of editing functions. All types of objects can be created, moved, deleted. In addition boxes can be resized, edges can be added or removed intermediate points (called "nails") for broken arrows, states can be declared initial and can be explored/unexplored (folded/unfolded).

There is no structural consistency requirement on edited objects. Only at translation into FC2 are such consistency rules checked.

3.5 Labeling and Annotating

All object types can be labeled. Following the FC2 syntactic conventions these labels are split in four distinct fields: `behav`, `struct`, `logic` and `hook` according to intention. Of course labeling is mostly optional. The *Label:Create/Edit All* menu button selects the full editor which is popped at each further mouse click on objects. There are four edition areas, corresponding to the four labeling fields above. As a shorthand the *Label:Create/Edit Default* menu button allows one-field edition, of `behav` labels for edges, webs and ports, of `struct` labels for vertices and boxes. This simpler function covers 90

Labels are displayed on the same drawing area as objects, which can be overwhelming sometimes. Other buttons from the *Label:* menu allow to hide or unmask labels globally or individually (or as a selection set), from specific labeling fields or indistinctly.

Finally the *Label:Show Label/Object* highlights the bindings from labels and objects to one another.

3.6 Automatic Placing

The *Placing:Explore* button allows to start or resume unfolding on states/vertices. States with incomplete display of outgoing transitions are identified by a smaller circle inside them. *Placing:Unexplore* allows to fold back states or transitions out of sight.

From the *Placing:Align* submenu sets of selected objects (right mouse button, remember?) can be aligned horizontally or vertically, from their centers, their left, right, upper or lower corners. They can also be projected on a circle: drag the mouse from the intended center to any point to lay on the circle itself.

Placing:Align:Spring calls an automatic layout algorithm called SPRING (courtesy of Michel Baudoin-Lafond, from LRI/Université d'Orsay), based on minimisation of a certain attraction/repulsion function amongst states.

3.7 Debug (diagnostic recovery)

Provided a diagnostic information (a path usually) was obtained in FC2 using the `-debug` option of FC2EXPLICIT or FC2IMPLICIT, it can be explored and mapped on the original distributed network representation. To do this, first load the `.fc2` file and explore it. Notice behaviours are now vectors of references. Then by selecting the *Debug:Edge/Vertex* mode, any click on an edge or vertex in this path will highlight corresponding elements in other windows containing the original network.

Warning: A number of assumptions are made here, for proper use. These are *not* checked by AUTOGRAPH and may result in error. First, it is supposed that all files comprised in the network are present, even if iconified windows. Second, AUTOGRAPH windows should not have been changed by edition (other than harmless small moves) since last translated into FC2 files. Third, the basename of the `basename.fc2` file *should be identical* to the `basename.atg` file in which the graphical description was stored.

In the current version the path is displayed both on transitions of individual automata , and on ports of boxes containing them. Vertices, edges, ports and the corresponding labels are highlighted. As the same component can occur several times in the description, boxes are assigned integer references and these integer are used everywhere to record to which box the behavioural element is tied. While this requires information deciphering to get used to, it was found preferable to the other option where windows were duplicated (to the point of submerging the screen ability for display).

3.8 Abstract Action

With this menu one can add annotation on an automaton to provide relevant informations so that it can be interpreted and translated as an abstract action.

The *AbstractAction:begin* menu button selects the abstract action initial state.

The *AbstractAction:end* menu opens a vertex as successful terminal state of an abstract action, whose name has to be provided then in a textual editor.

The *AbstractAction:save* translates the window content in fc2 format as an abstract action. The net contains a hook "abstract_action", the begin state have a logic "initial" and the end state have a behav giving the name of the abstract action.

3.9 Translation into FC2

Translation from graphical representations to FC2 files is quite straightforward, specially on automata. There is a number of consistency checks to insure safe interpretation (in fact just common sense considerations):

- Automata *must* have an initial state;
- Boxes *may not* overlap (proper nesting);
- Innermost boxes must have all their ports labeled, and contain either a `struct` name (the subcomponent to be instantiated later from another source description) or an automaton;
- Edges should not link a vertex to a port/web, and not two ports apart from neighbouring boxes (siblings or "mother/daughter" in the containment tree).
- Connections should not contain more than one external port (without external port, the connection is called *internal* to the subnetwork represented by the mother box, and correspond to an action hidden at this level).

Connections here are sets of ports bound together by being linked to the same webs (so the FC2 format allows multipoint synchronisation). As a shorthand two ports can be directly linked by an edge for a binary synchronisation. Each connection will produce a synchronisation vector describing a possible behaviour of the (subnetwork translated from their) mother box. Synchronisation vectors will be labeled (or internal) according to the external port of the connections.

Globally visible actions are formed by outermost webs, ports and edges bearing an explicit label (a box is said to be *outermost* if not nested inside another one, *outermost* ports are ports on outermost boxes, and outermost webs/edges are tied only to outermost ports).

The previous example from section 2.4 already showed ATG drawings and their FC2 counterpart.

4 The FC2 file linker FC2LINK

A complete network description may be split amongst several actual files, possibly originated from different sources, textual or graphical. This allows components reuse and modularity. On the other hand most verification tools will only accept a single file input. Linking files together consists mainly in ensuring a proper correspondence in label references, between the locations where subcomponents are defined and their invocation in a larger network. Example of this is provided in figure 5, where the *fork* description in figure 3 is substituted to its reference inside previous network of figure 4. Tabular references must be merged, and so usually shifted to avoid conflicts.

FC2LINK requires a `-main` filename, whose topmost network (`net0`) will be taken to become the global network. Hierarchical subcomponents are only selected from the set of FC2 files provided as arguments as they are needed, through dependency analysis. Ambiguity results in errors.

5 Global System Generation

The global model construction/expansion is a main part of model-based verification tools. States in such a model are vectors of component (local) states, and behavioural transitions are obtained by interleaving or synchronization of local behaviours. Of course this means potential combinatorial explosion, and methods for compact representation of global state spaces are at the core of all approaches to model-based verification techniques.

FCTOOLS offers two alternative implementations of the product construction: `fc2glob`, classically based on enumerated representation of states and transitions; `fc2iglob`, a symbolic version based on *Binary Decision Diagrams* for implicit representation of (sets of) states. Both are embedded in the respective commands `fc2explicit` and `fc2implicit`.

While the explicit product construction yields naturally a full automaton (with transitions), the implicit *BDD* implementation produces rather a symbolic version of the global reachable state space, so that producing a full global FC2 automaton requires more effort to list transitions for file printing. On the other hand many subsequent analysis do *not* require the actual automaton at this stage.

5.1 The Explicit Global System Generator FC2GLOB

The construction algorithm is rather straightforward. Hash tables are used to keep the set of already reached states represented as bit vectors, and new discovered states are given an integer reference and stored in a list of "states to explore". Target states are stored as part of the source state description together with the labeled transition reaching them.

When invoked recursively on a multi-level hierarchical network the explicit implementation can be alternated with reduction functions at intermediate stages with the `-comp` option. One recovers then the *compositional model reduction* approach popularized through the original AUTO tool.

Synchronisation vectors can be applied in any of two ways. When the `-bitset` flag is on, a bitvector mask selects applicability on any (bitvector) state, and other bitvector functions then actually compute the target state. The other way is more traditional. It was found experimentally that the `bitset` approach works better for large vectors with components of few states (the uncompositional, flat approach), while traditional transition application retains efficiency when components put in parallel were themselves large automata.

5.2 The Implicit Global System Generator FC2IGLOB

FC2IGLOB (or `fc2implicit -reach`) computes the (BDD characteristic formula given an ordered boolean encoding of) the set of global reachable states of the system. No compositional speed-up method is foreseen, so that the network is flattened to a single-level vector of individual automata. The reachable state space is of course evaluated in a breadth first search strategy, applying event synchronisation vectors individually until fixpoint. This gains efficiency as the symbolic representation of a given synchronisation vector does not deal with idle components.

Fixpoint reachable state computation can be refined to allow for on-line deadlock detection (states without behaviours), and followed by livelock or divergent states detection on the result (a divergent state may perform infinite sequences of hidden "tau" actions, a livelock state can exhibit *only* such behaviours).

The tool only enumerates states if asked to produce the FC2 automaton on file with the `-fc2` option (otherwise it provides size figures). If deadlock/livelock/divergent states were queried and found, it provides a diagnostic path. If `-debug` option is used, additional information is inserted about the origin of transitions in terms of network components.

6 Bisimulation minimisation and equivalence checking

These functionalities are implemented both with implicit and explicit representation technologies. In the former case the algorithm assumes a network description as argument (and *not* a single automaton, so as to benefit from cleaver encoding using boolean variables); in the latter case a global automaton is built prior to minimisation, but compositional reductions can be applied on hierarchical network descriptions. Experience showed that explicit methods can run substantially faster when the size of the considered automaton is still manageable for them. On the other hand symbolic methods are sometimes applicable on large systems, provided the number of classes remain low (for instance in *weak* bisimulation when only a few signals are left visible to distinguish between states). Also they have a clear use when only comparing two distinct networks (the *equivalence checking problem*).

The tools deal with all three standard variants, namely strong, weak and branching bisimulation. The *Relational Coarsest Partitioning Algorithm* of Kanellakis and Smolka [2] is used to refine a partition of the states, until fixpoint. In case of *equivalence checking* of two distinct automata the refinement can possibly be aborted before fixpoint, when it is found that some state has no match left from the other automaton.

6.1 The Implicit Algorithm

Symbolic algorithms for the computation of (strong, weak or branching) bisimulation equivalence classes were described in [1].

The quotient automaton can be produced in FC2 through symbolic projection functions, to the effect of replacing any (symbolic) state by a uniquely determined representative, and then providing integer representations of such representative to be used as new target states.

When checking for equivalence between two distinct networks the synchronous product is built so that only couple of states reached in some common behavioural path are tried for bisimulation. This instills some "on-the-fly" flavor to the approach. A path leading to a matchless state with minimal splitting iteration index is produced when debug flag is on.

See section 2.3 for UNIX command syntax.

6.2 The Explicit Algorithm

Can be iteratively applied on automata resulting from subprocesses for compositional reductions (using the `-comp` flag). Builds a global automaton,then reduces it into another explicit automaton, minimal in states and with only transitions explicit in the former ones (no τ transitive closure in case of observational bisimulation).

Warning: under development. When checking for equivalence between two distinct networks the disjoint union of the two state spaces is built, and then partitioned as a whole. The algorithm then possibly aborts because a class contains no states from one of the automata, *before* reaching fixpoint. Then a list of states without match is provided as counterexample.

See section 2.3 for UNIX command syntax.

7 The Model Abstraction

Abstract Actions allow to observe an automaton with a coarser atomicity level. So it appears as the opposite of refinement, and as such takes an important role in analysis, where preservation of some prior less detailed version can be a useful check. The idea is to collapse a number of sequences of concrete behaviours as "abstractly equivalent" and atomic, calling such a set an abstract action. Any concrete behaviour sequence somehow "implements" the abstract behaviour then. For finiteness reasons and efficient automatic verification we restrict to the case where abstract actions correspond to *regular expressions* of (alternative) concrete behaviours, and thus to finite automata.

Reducing a global system with respect to a set of abstract actions results in a system conceptually simpler, where meaningful activities have been isolated as coarser-grain atoms and named. A practical subcase arises when a *single* abstract action consisting of the complement of the desired behavioural trace language is provided. Then finitary trace inclusion is obtained by checking the non admission of the abstract action by the system. In any case, *determinism* is a desirable property for abstract actions, concerning algorithmic

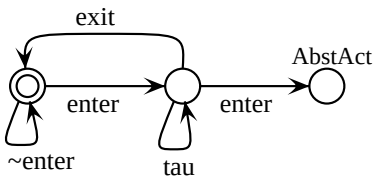


Figure 9: philosophers abstract-action

efficiency. In general abstraction is *not* compositional w.r.t. network operators such as parallelism.

Abstract actions are gathered in a new alphabet, to be labels of new transitions. New states are based on concrete ones (only that some may usually become unreachable). We currently input abstract actions as automata in the FC2 format, using the following syntax to represent sequence of concrete actions:

$$\begin{aligned} single-action &= ID|?ID|\#ID|!ID|\star \\ abstract-action &= \sim single-action|single-action.abstract-action \end{aligned}$$

$\star$ is the "true" wildcard action and and represents any concrete action while the "false" action is $\sim\star$. To match any path that contains the concrete action `?a.#b.!c`, we have to provide in the abstract action automaton a transition labeled by `?a.#b.!c.`$\star$.

For instance, in figure 9 we use the ATG abstract-action feature to describe an abstract behaviour refuting mutual exclusion.

The fc2 description below corresponds to the translated form of the figure 9.

```
nets 1
hook"main" > 0
struct"AbstAct"
net 0
behavs 3
:0 "exit"
:1 "enter"
:2 "AbstAct"
logic "initial">0
hook "abstract_action"
vertice 3
vertex0
edges 2
edge0
behav ~1
 -> 0
edge1
```

```
behav 1
 -> 1
vertex1
edges 3
edge0
behav tau
 -> 1
edge1
behav 1
 -> 2
edge2
behav 0
 -> 0
vertex2
behav 2
```

7.1 The Explicit Abstractor FC2ABST

From the description above each abstract action is in essence an automaton with transition labeled by (expressions on) concrete behaviours, with a dedicated terminal state itself labeled by the abstract name. Then a criterion is a collection of such automata, to be gathered in a single one by classical union.

The abstracted automaton is built by constructing some synchronous product of the network with this structure, and setting new states and transitions in the result only when terminal criteria states are reached (bearing their name to become the new transition label, while the target state correspond to the one produced from the network alone). This procedure has to be applied for each new created state facing the whole criterion.

7.2 The Implicit Abstractor FC2IABST

Warning: under development. From the transition relation of the global automaton and the abstraction criterion, an abstract transition relation is built. Then, to get the abstract model, we compute the reachable states from the initial state with the new transition relation. The command `fc2iabst` is actually a restricted use of the tool command `fc2implicit`. One has in fact to give two FC2 files as input to the command, the first being the network description and the second the abstract criterion. Result output option is automatically set. See section 2.3 for UNIX command syntax.

8 Verification by Observers

A great deal of practical verification is usually conducted by compiling an automaton-like structure from the property to establish, with possibly additional annotations on states and transitions of various sorts (*success*, *failure* or *recur* states, *don't care* transitions,...).

Verification then starts by constructing a synchronised product of the (usually large) network state space with the (usually smaller) state space of the observer structure. One can attempt to introduce the actual verification algorithms in the middle of this construction, to get potential negative results as early as possible (known as "on the fly" or "local" techniques).

Here again the distinction between implementations based on explicit and implicit state representation are relevant. Symbolic techniques are usually a clear winner, specially when no representation of subsets of transitions are required, and only forward search across states is needed (since backward search may exit the reachable state space and needs to be controled). This is the case for *safety* properties.

In the current version of FC2IMPLICIT one can only specify *deadlocks*, *livelocks*, and *divergent* states as particular configurations. A dealock is a state without outgoing transitions, a livelock is a state from which there will never eventually be produced any visible action, a divergent state is one from which there is an infinite behavioural sequence without visible label content. In practice what this means is that special recognizing states in observers should be deadlocks, inducing deadlocks in product machines also, for safety properties. For *liveness* properties the unwanted nonprogress loops should correspond to hidden behaviours. In all case this is awkward and the current situation is not as expressive as should be. We plan to extend this with far more flexible descriptions of particular states and transitions as the FC2 format made special provision for that.

The combined construction poses little problem and can actually be described *inside* the formalism, by setting the network and the observer in "regular" parallel. According to flag options selecting which particular feature is looked for, one discovers symbolically these states/loops from the network which can be coupled (in the synchronous product) to particular states of the observers. Then a (shortest) path from initial state to one of the state identified as such is produced. Finally, if `-debug` option is set, source recovery functions query the state structural fields to uplift this diagnostic back to the original multifile network description, and to AUTOGRAPH display.

9 Source Information Recovery

When invoked with -`debug` option, both FC2IMPLICIT and FC2EXPLICIT preserve a structural correspondence between states: a global state obtained by product is a (comma separated) list of local states in the innermost components of the network from which the global automaton originated ;a reduced state in a quotient automaton is a union/sum of states in the unreduced former automaton. In case of compositional alternated products and reductions one gets corresponding alternation of parenthesized sum and list expressions. In addition the FC2 files contains in its header a description of the current structure (in terms of files where constitutive elements were found).

The previous elements allow at any time to retrieve how states distribute on the original descriptions. No such information is kept for transition labels, but the mapping of global behaviours into local ones, once known the source and target states, is usually straightforward (and if several cases apply, they are all valid anyway). Such a reconstruction is performed after observational minimisation in the -`debug`case. It should be noted that preservation of debug information can be space and time consuming, which is why it is turned on only on explicit flag option.

When -`fc2`and -`debug` option flags are set, potential counterexample paths are completed with transition behaviour label mapping down to components. In this way the counterexample files can be loaded to AUTOGRAPH or FC2VIEW for display on distributed networks, where local state changes and behaviours are highlighted. See further description in AUTOGRAPH and FC2VIEW sections.

References

[1] A. Bouali and R. de Simone. Symbolic bisimulation minimisation. In *Fourth Workshop on Computer-Aided Verification*, volume 663 of *LNCS*, pages 96–108, Montreal, 1992. Springer-Verlag.

[2] P.C. Kanellakis and S.A. Smolka. CCS expressions, finite state processes, and three problems of equivalence. *Information and Computation*, 86:43–68, 1990.

[3] E. Madelaine and R. De Simone. The FC2 Reference Manual. Technical report, INRIA, 1993. available by ftp from `cma.cma.fr:pub/verif` as file *fc2refman.ps.gz*.

[4] J.K. Ousterhout. *Tcl and the Tk Toolkit*. Professional Computing Series. Addison-Wesley, 1994.

Unité de recherche INRIA Lorraine, Technopôle de Nancy-Brabois, Campus scientifique,
615 rue du Jardin Botanique, BP 101, 54600 VILLERS LÈS NANCY
Unité de recherche INRIA Rennes, Irisa, Campus universitaire de Beaulieu, 35042 RENNES Cedex
Unité de recherche INRIA Rhône-Alpes, 46 avenue Félix Viallet, 38031 GRENOBLE Cedex 1
Unité de recherche INRIA Rocquencourt, Domaine de Voluceau, Rocquencourt, BP 105, 78153 LE CHESNAY Cedex
Unité de recherche INRIA Sophia-Antipolis, 2004 route des Lucioles, BP 93, 06902 SOPHIA-ANTIPOLIS Cedex

Éditeur
INRIA, Domaine de Voluceau, Rocquencourt, BP 105, 78153 LE CHESNAY Cedex (France)
ISSN 0249-6399